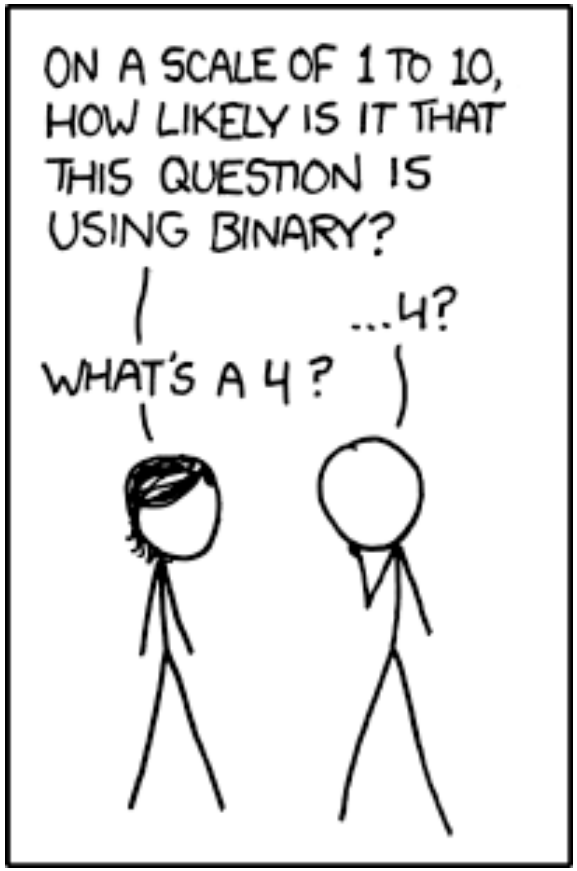
Memory, Data, & Addressing I

CSE 351, Winter 2023

Instructor:
Sam Wolfson

Teaching Assistants:
Aman Mohammed
Angela Xu
Armin Magness
Clare Edmonds
David Dai
Jenny Peng
Maggie Jiang
Mara Kirdani-Ryan
Nayha Auradkar
Yoonseo Song



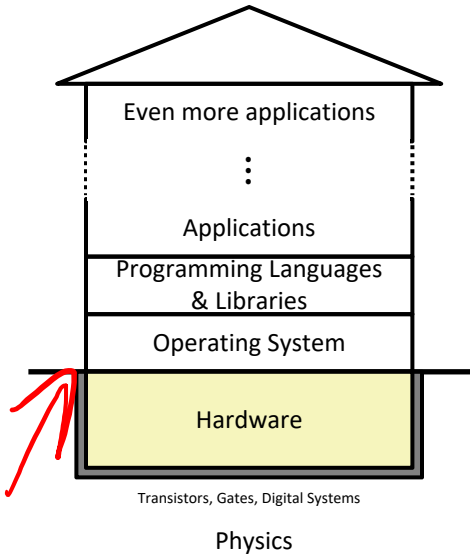
http://xkcd.com/953/

Relevant Course Information

- Everything except readings due @ 11:59 pm
  - Course Policies and Pre-Course Survey due tonight
  - Binary Homework due Monday (1/9)
  - Lab 0 now due **Wednesday** (1/11)
    - This lab is *exploratory* and looks like a homework; the other labs will look a lot different
- Ed Discussion etiquette
  - For anything that doesn't involve sensitive information or a solution, post publicly (you can post anonymously!)
  - If you feel that your question has been sufficiently answered, make sure that a response has a checkmark ✅
- Everyone should have accounts on attu now
- Please leave space at the edges of rows for folks arriving late
  - This building is far away from the Gates lecture halls!

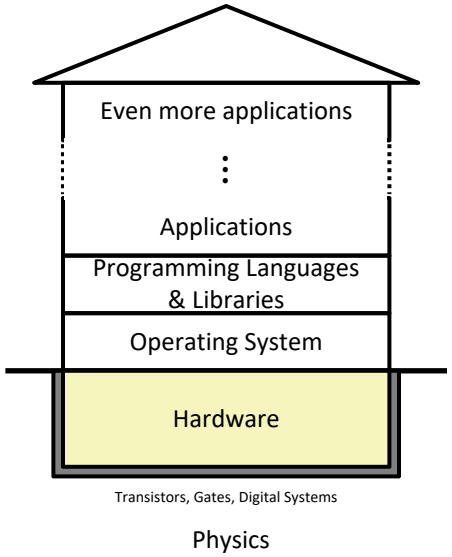
The Hardware/Software Interface

- Topic Group 1: **Data**
  - **Memory, Data**, Integers, Floating Point, Arrays, Structs
- Topic Group 2: **Programs**
  - x86-64 Assembly, Procedures, Stacks, Executables
- Topic Group 3: **Scale & Coherence**
  - Caches, Processes, Virtual Memory, Memory Allocation



The Hardware/Software Interface

- Topic Group 1: **Data**
  - **Memory, Data**, Integers, Floating Point, Arrays, Structs

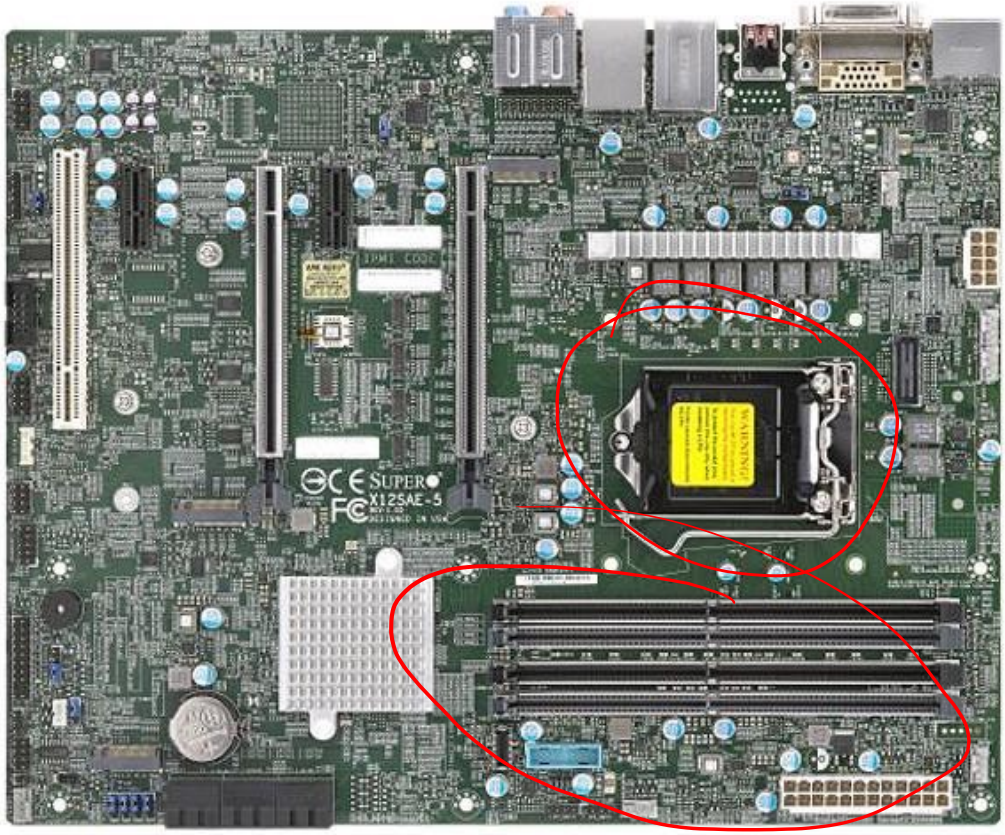


- How do we store information for other parts of the house of computing to access?
  - How do we represent data and what limitations exist?
  - What design decisions and priorities went into these encodings?

Hardware: Physical View

Bus connections

USB…



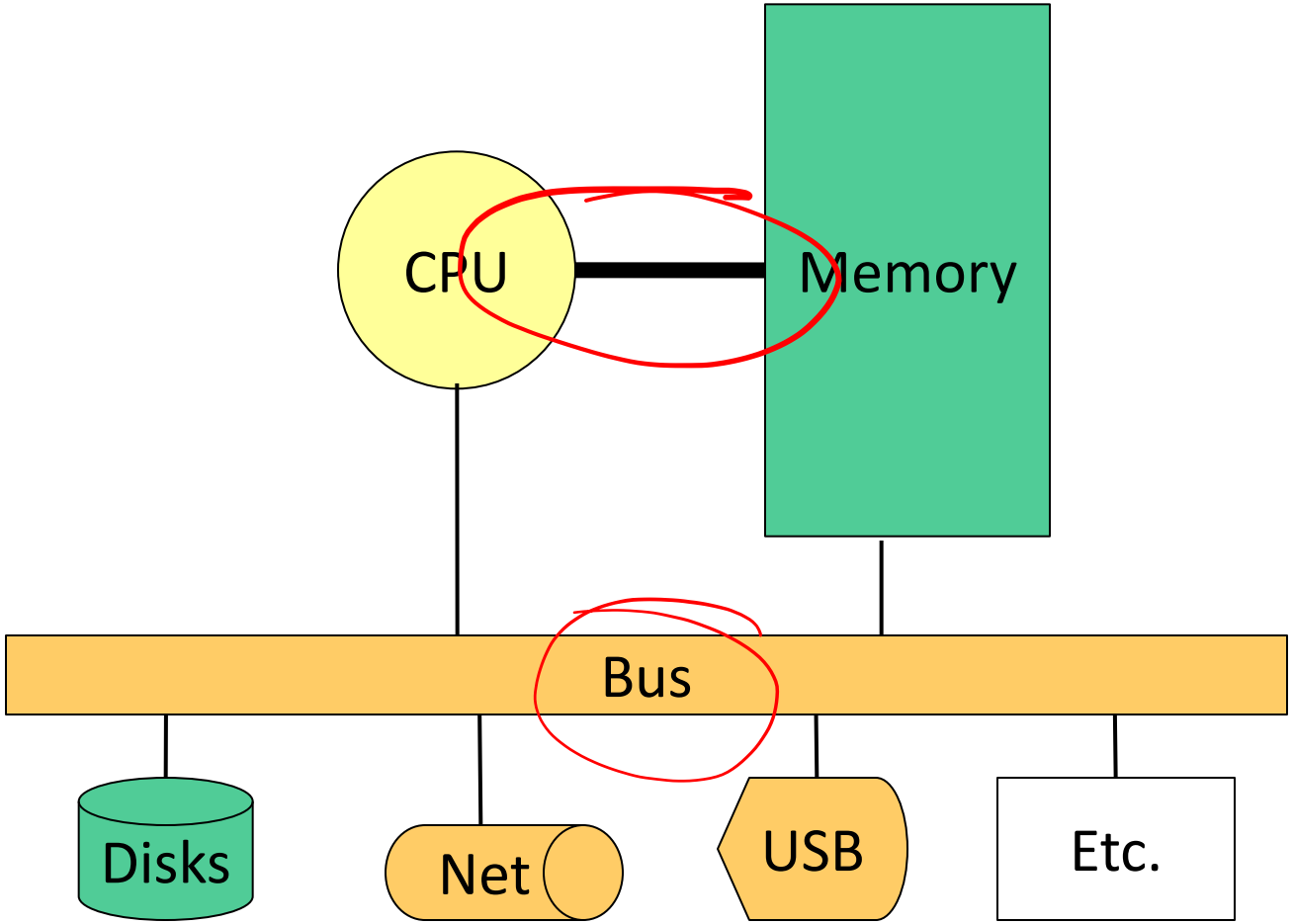
CPU
(empty slot)

Memory

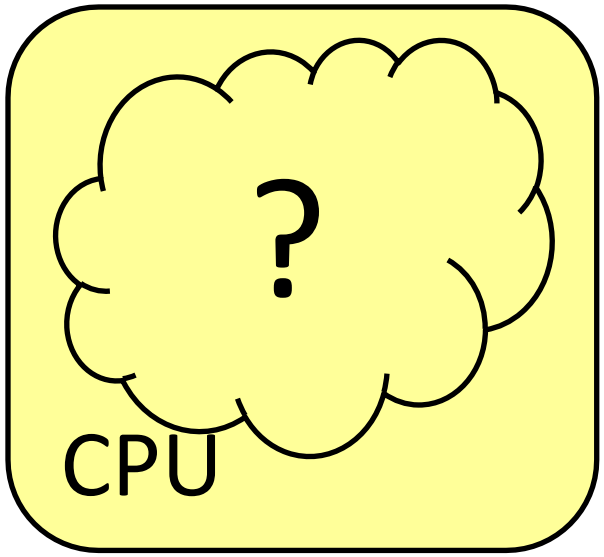
I/O
controller

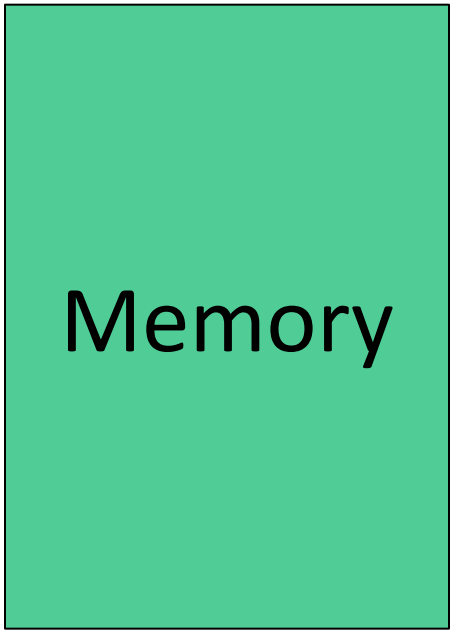
Storage connections

Hardware: Logical View



Hardware: 351 View (version 0)





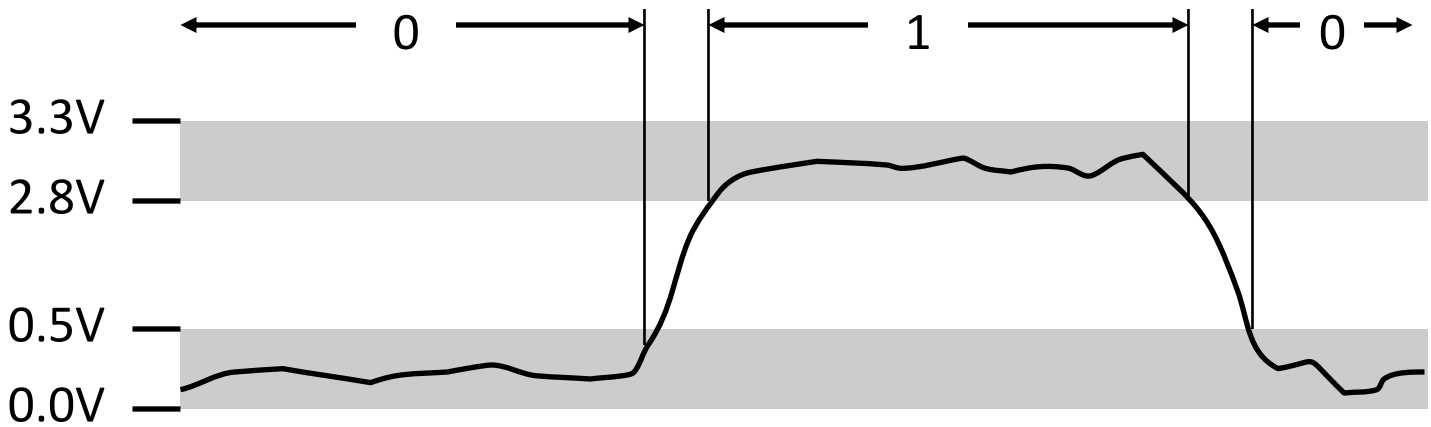
- The CPU executes instructions
- Memory stores data

Q1: How are data and instructions represented?

- Binary encoding!
  - Instructions *are* just data

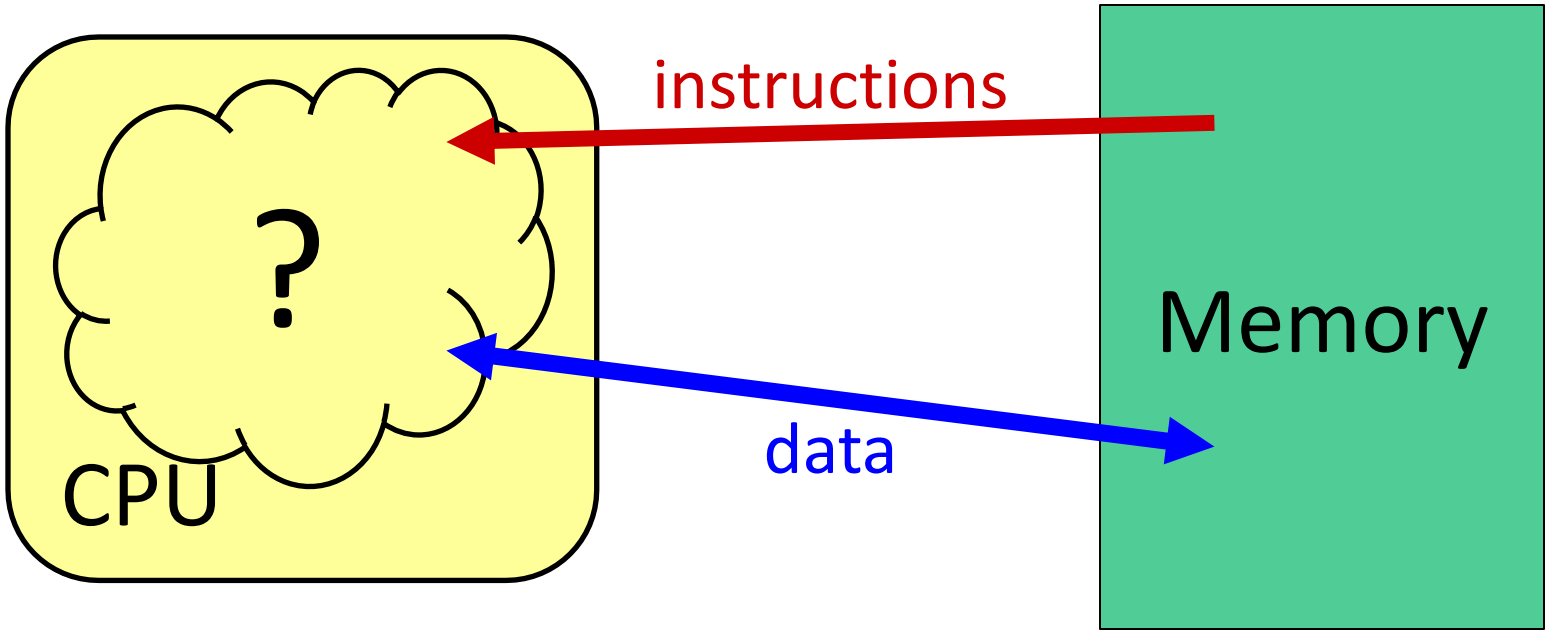
Aside: Why Base 2?

- Electronic implementation
  - Easy to store with bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires



- Other bases possible, but not yet viable:
  - DNA data storage (base 4: A, C, G, T) is hot @UW
  - Quantum computing

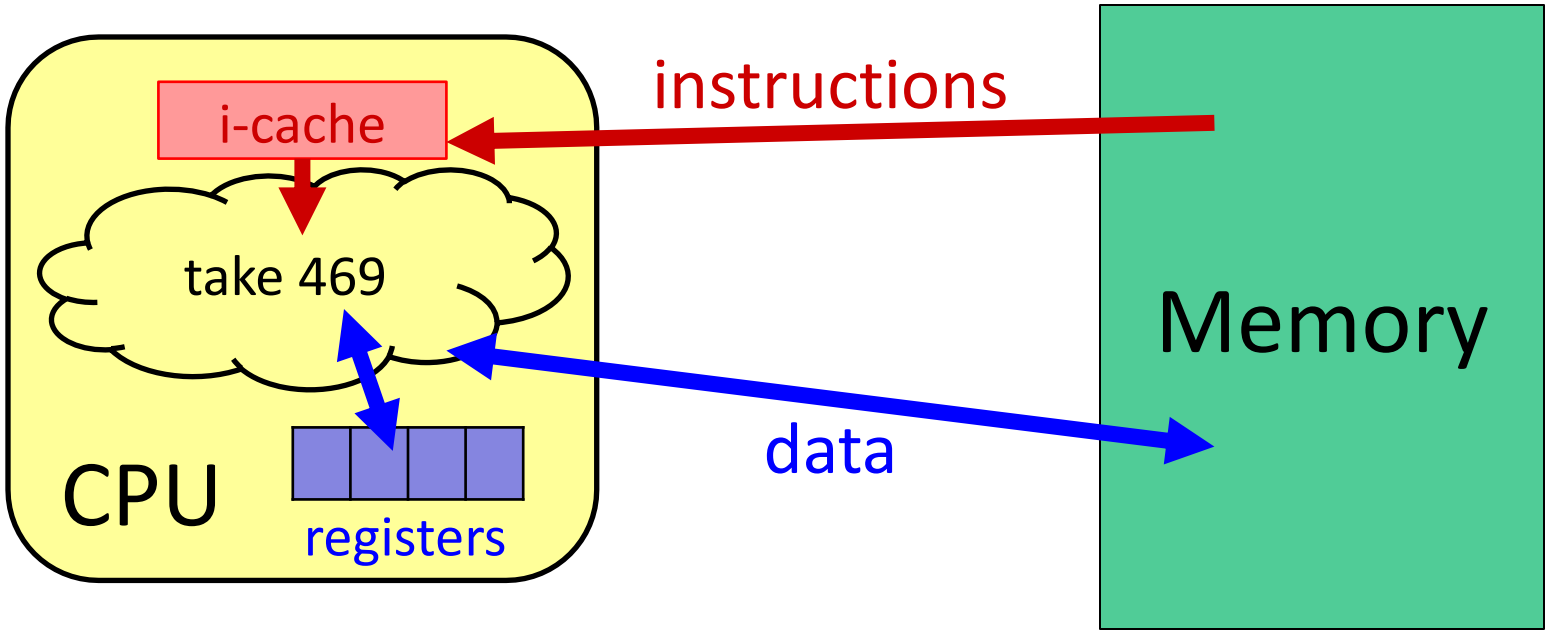
Hardware: 351 View (version 0)



- To execute an instruction, the CPU must:
  1) Fetch the instruction
  2) (if applicable) Fetch data needed by the instruction
  3) Perform the specified computation
  4) (if applicable) Write the result back to memory

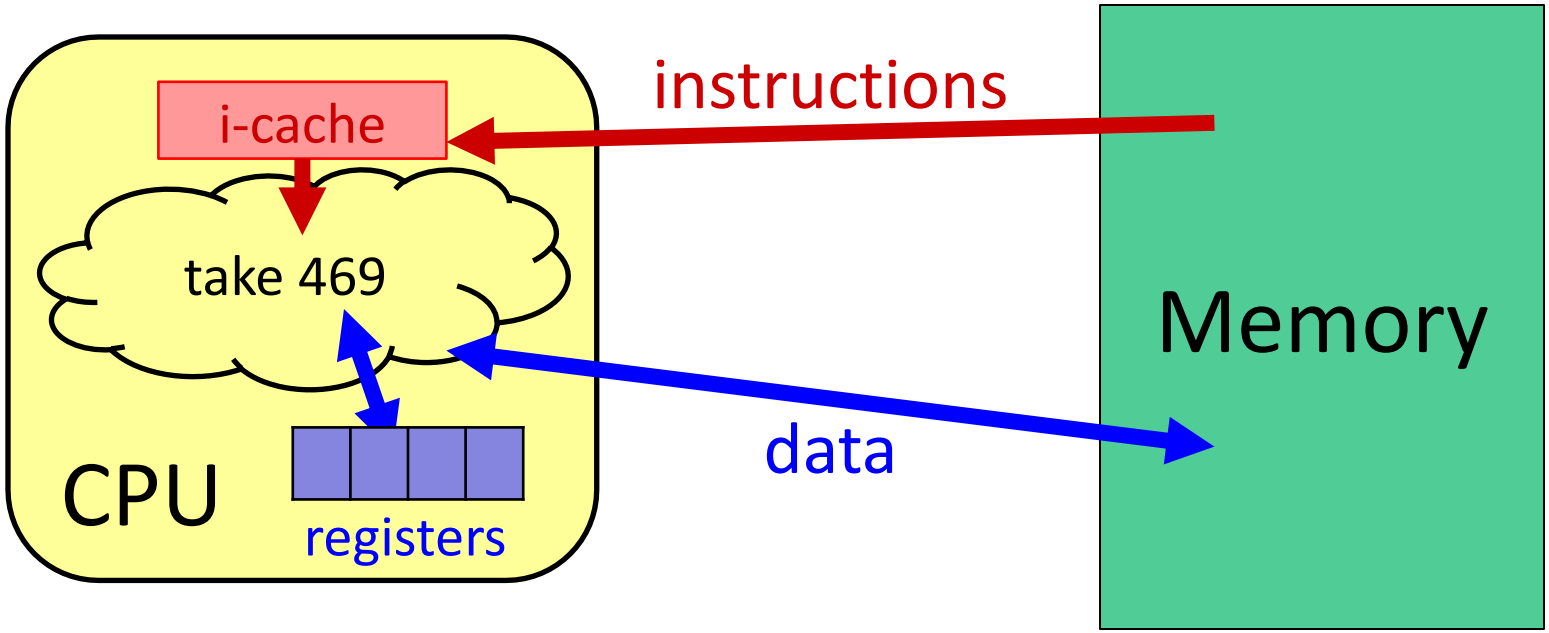
Hardware: 351 View (version 1)

This is extra (non-testable) material



- More CPU details:
  - Instructions are held temporarily in the instruction cache
  - Other data are held temporarily in registers
- Instruction fetching is hardware-controlled
- Data movement is programmer-controlled (assembly)

Hardware: 351 View (version 1)



- We will start by learning about Memory

Q2: How does a program find its data in memory?

- Addresses!
  - Can be stored in *pointers*

Reading Review

- Terminology:
  - word size, byte-oriented memory
  - address, address space
  - most-significant bit (MSB), least-significant bit (LSB)
  - big-endian, little-endian
  - pointer

- Questions from the Reading?

Review Questions

- By looking at the bits stored in memory, I can tell what a particular 4 bytes is being used to represent.

  A. True **B. False**

- To fetch a piece of data from memory, all we need to know is its address.

  A. True **B. False**

- Which of the following bytes have a most-significant bit (MSB) of 1?

  A. 0x63 **B. 0x90** **C. 0xCA** **D. 0xF**

  0101 1001 1100 0x0F

Fixed-Length Binary (Review)

- Because storage is **finite**, everything is stored as "fixed" length
  - Data is moved and manipulated in fixed-length chunks
  - Multiple fixed lengths (*e.g.*, 1 byte, 4 bytes, 8 bytes)
  - Leading zeros now *must* be included up to "fill out" the fixed length

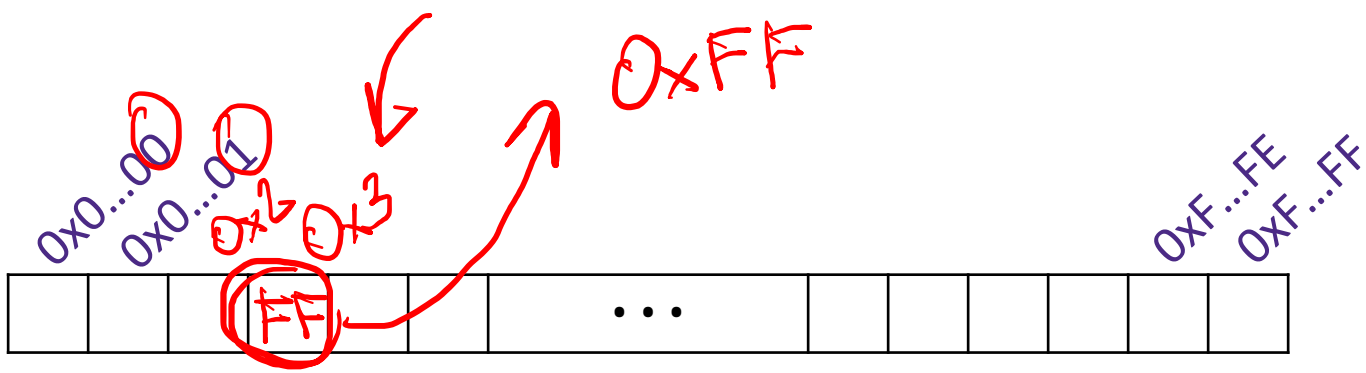
- Example: the "eight-bit" representation of the number 4 is 0b00000100

Most Significant Bit (MSB)

Least Significant Bit (LSB)

Bits and Bytes and Things (Review)

- 1 byte = 8 bits
- n bits can represent up to 2^n things
  - Sometimes (oftentimes?) those "things" **are also bytes**!
- If addresses are a-bits wide, how many distinct addresses are there?
- What does each address refer to?



Machine "Words" (Review)

- Instructions encoded into machine code (0's and 1's)
  - Historically (still true in some assembly languages), all instructions were exactly the size of a word
- We have *chosen* to tie word size to address size/width
  - word size = address size = register size
  - word size = w bits → 2^w addresses
- Current x86 systems use **64-bit (8-byte) words**
  - Potential address space: $\mathbf{2^{64}}$ addresses
    2^{64} bytes ≈ **1.8×10^{19} bytes**
    = 18 billion billion bytes = 18 EB (exabytes)
  - Actual physical address space: **48 bits**

- **TL;DR: when we say "word" in this class, we mean an 8-byte chunk of memory (the size of an address)**

Data Representations

- Sizes of data types (in bytes)

Java Data Type	C Data Type	IA-32 (old)	x86-64
boolean	bool	1	1
byte	char	1	1
char		2	2
short	short int	2	2
int	int	4	4
float	float	4	4
	long int	4	8
double	double	8	8
long	long long	8	8
	long double	8	16
(reference)	**pointer ***	**4**	**8**

address size = word size

To use "`bool`" in C, you must `#include <stdbool.h>`

Discussion Question

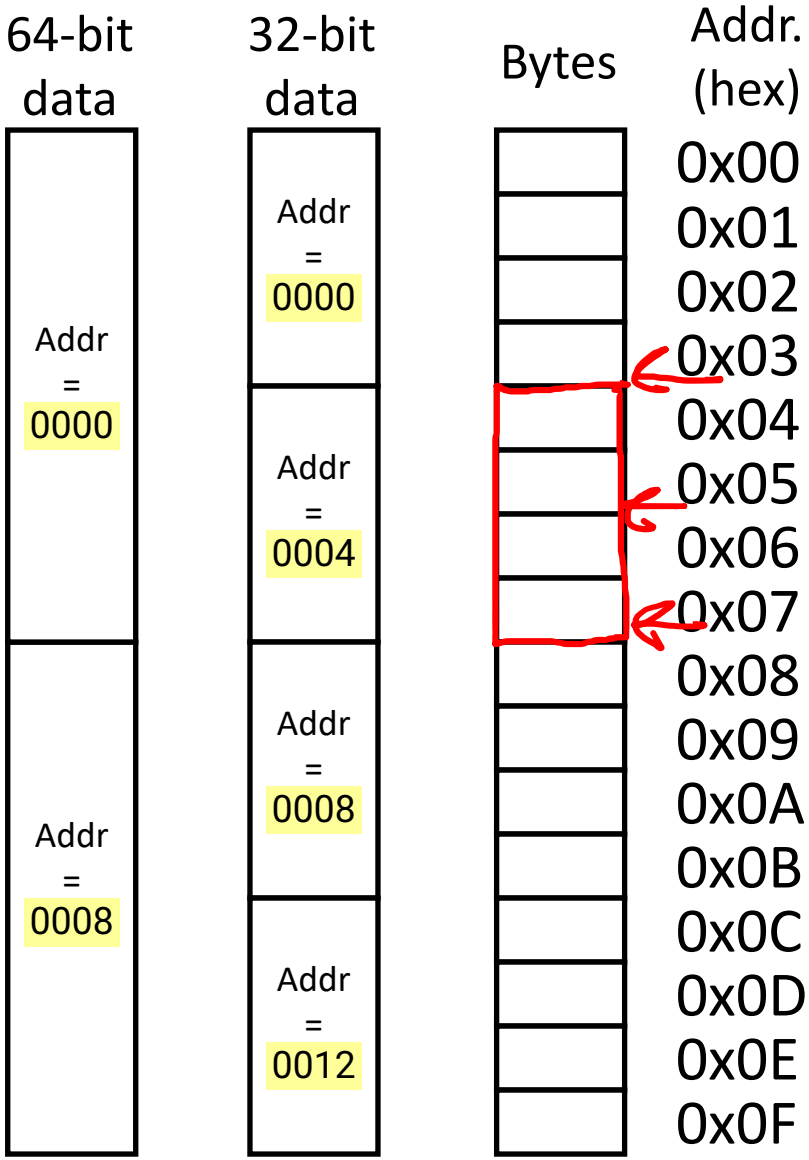
- Over time, computers have grown in word size:

Word size	Instruction Set Architecture	First? Intel CPU	Year Introduced
8-bit	??? (Poor & Pyle)	Intel 8008	1972
16-bit	x86	Intel 8086	1978
32-bit	IA-32	Intel 386	1985
64-bit	IA-64	Itanium (Merced)	2001
64-bit	x86-64	Xeon (Nocona)	2004

  - What do you think were some of the *causes*, *advantages*, and *disadvantages* of this trend?

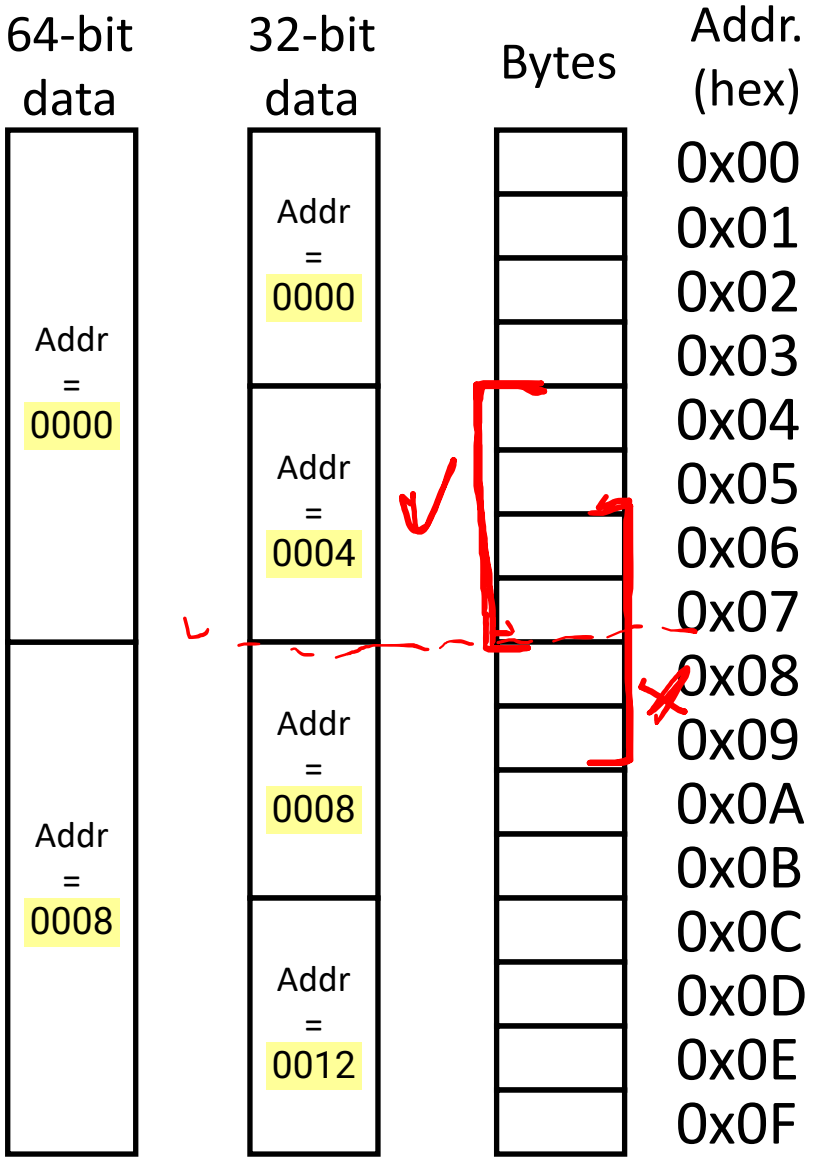
Address of Multibyte Data (Review)

- Addresses still specify locations of bytes in memory, but we can choose to *view* memory as a series of chunks of fixed-sized data instead
  - Addresses of successive chunks differ by data size
  - Which byte's address should we use for each word?
- The address of *any* chunk of memory is given by the address of the **first byte**
  - To specify a chunk of memory, need *both* its **address** and its **size**



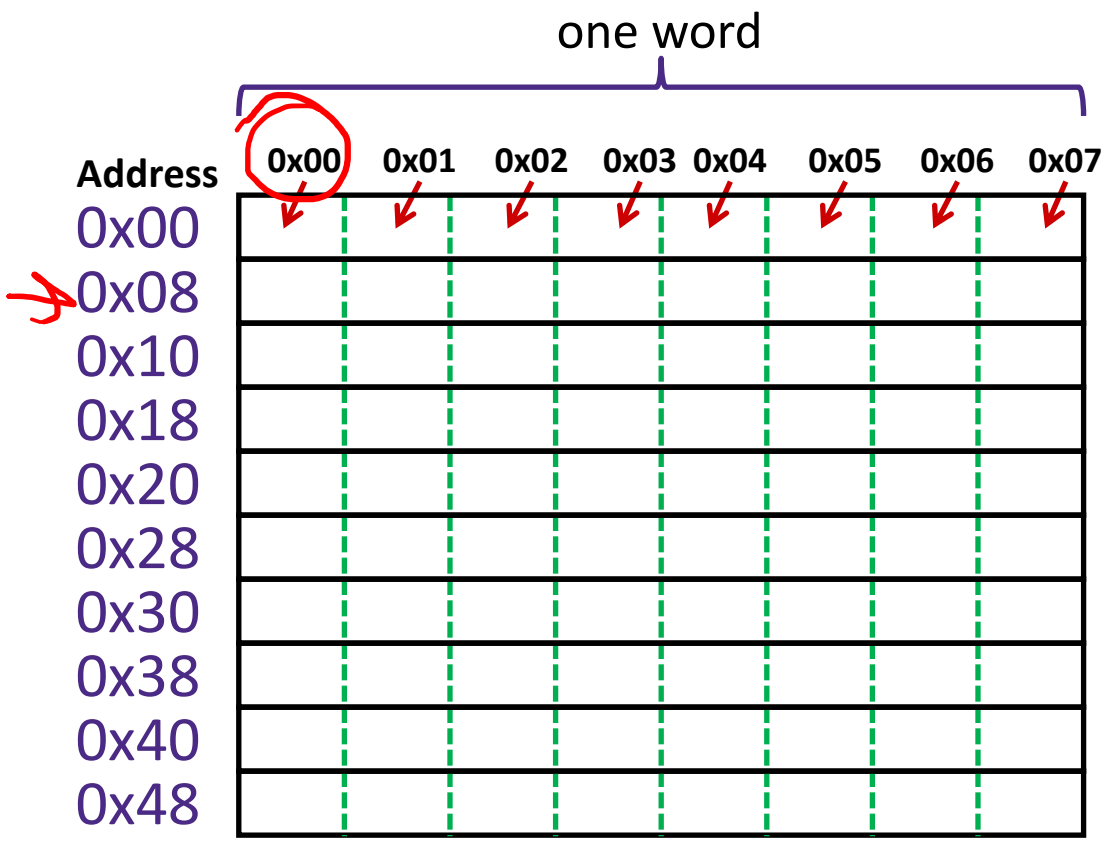
Alignment

- The address of a chunk of memory is considered aligned if its address is a multiple of its size
  - View memory as a series of consecutive chunks of this particular size and see if your chunk doesn't cross a boundary



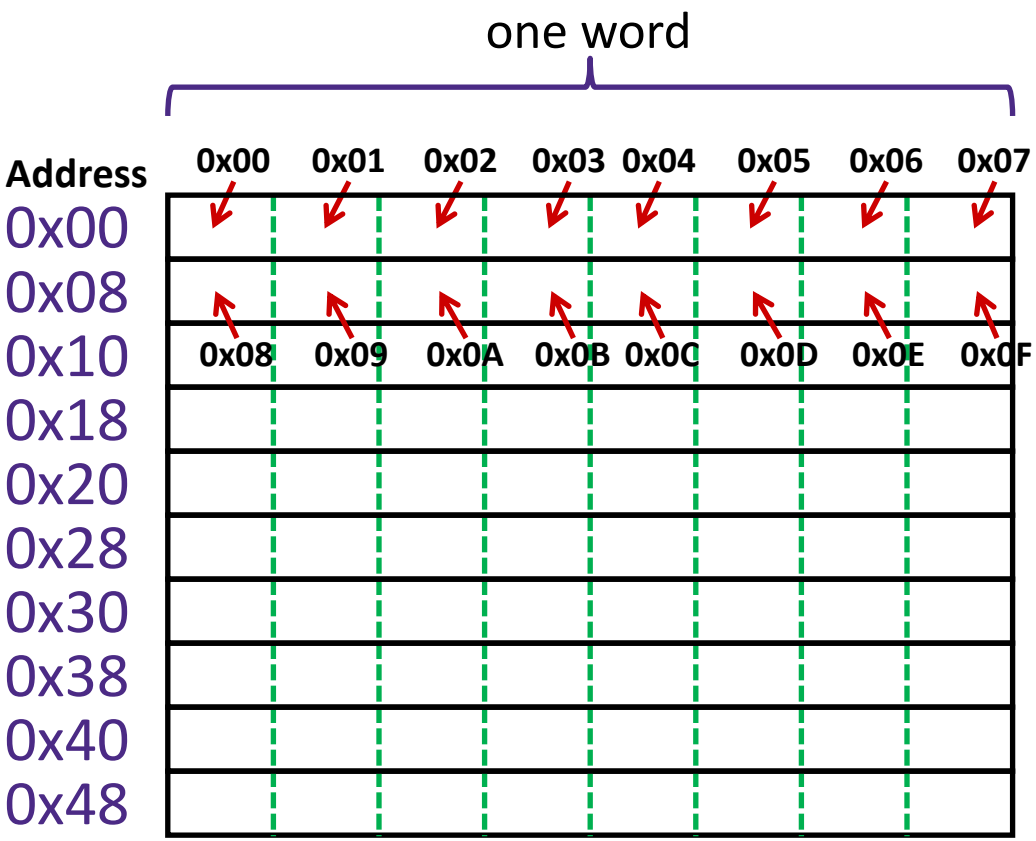
A Picture of Memory (64-bit view)

- A "64-bit (8-byte) word-aligned" view of memory:
  - In this type of picture, each row is composed of 8 bytes
  - Each cell is a byte
  - An aligned, 64-bit chunk of data will fit on one row



A Picture of Memory (64-bit view)

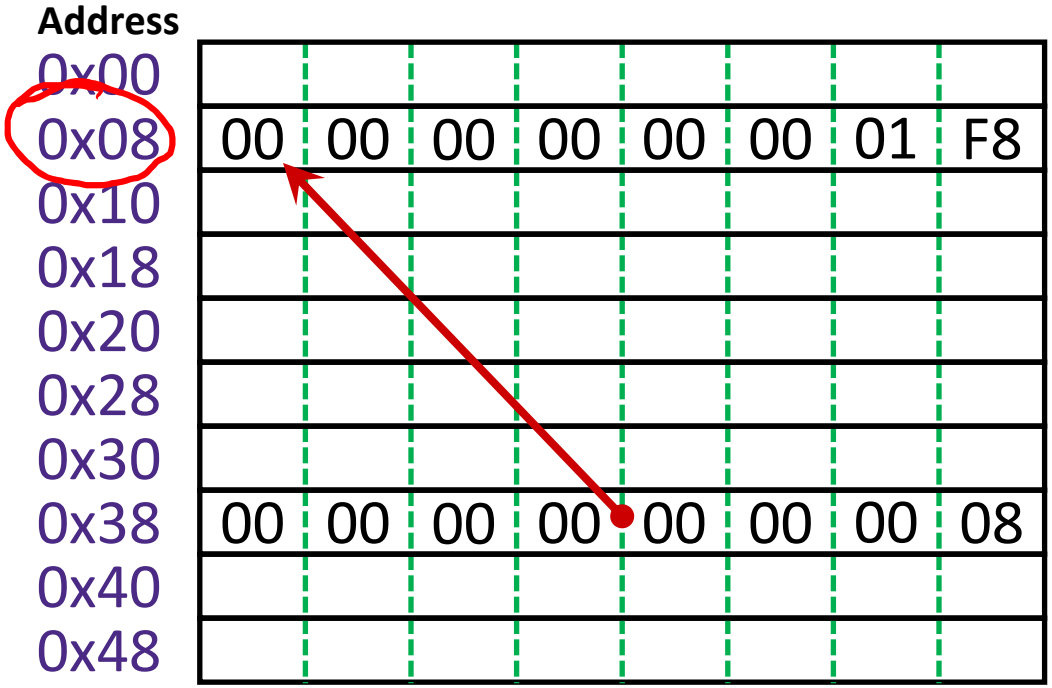
- A "64-bit (8-byte) word-aligned" <u>view</u> of memory:
  - In this type of picture, each row is composed of 8 bytes
  - Each cell is a byte
  - An aligned, 64-bit chunk of data will fit on one row



Addresses and Pointers

64-bit example
(pointers are 64-bits wide)
big-endian

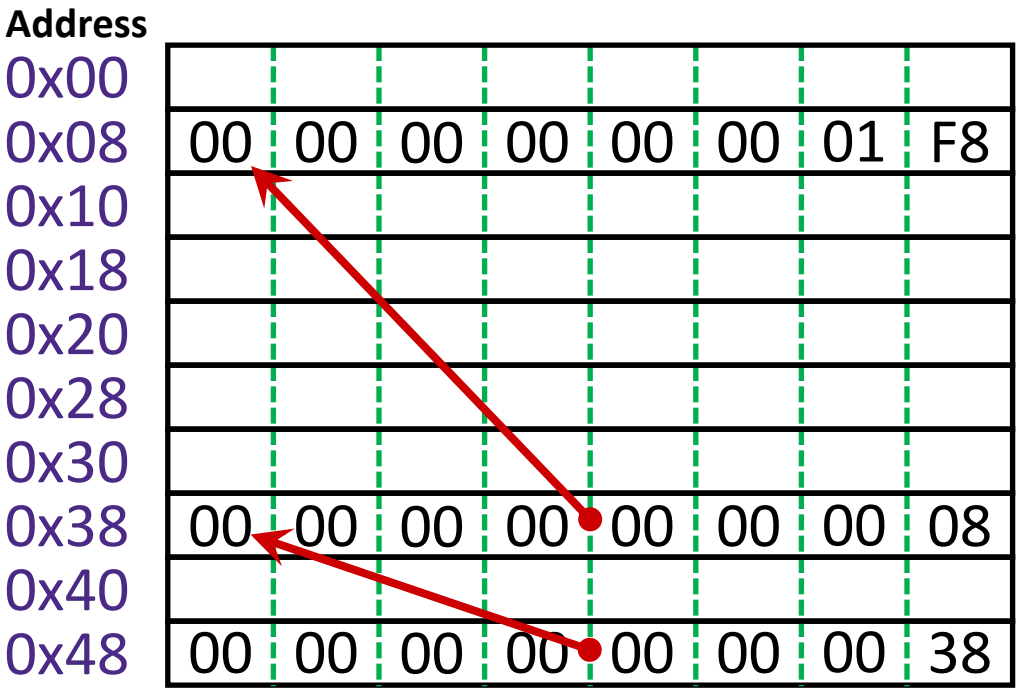
- An *address* refers to a location in memory
- A *pointer* is a data object that holds an address
  - Address can point to *any* data
- 8-byte value 504 stored at address 0x08
  - $504_{10} = 1F8_{16}$
    = 0x 00 … 00 01 F8
- Pointer stored at 0x38 points to address 0x08



Addresses and Pointers

64-bit example
(pointers are 64-bits wide)
big-endian

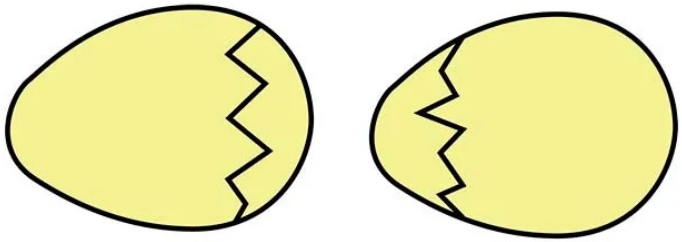
- An *address* refers to a location in memory
- A *pointer* is a data object that holds an address
  - Address can point to *any* data
- Pointer stored at 0x48 points to address 0x38
  - Pointer to a pointer!
- Is the data stored at 0x08 a pointer?
  - Could be, depending on how you use it



Byte Ordering (Review)

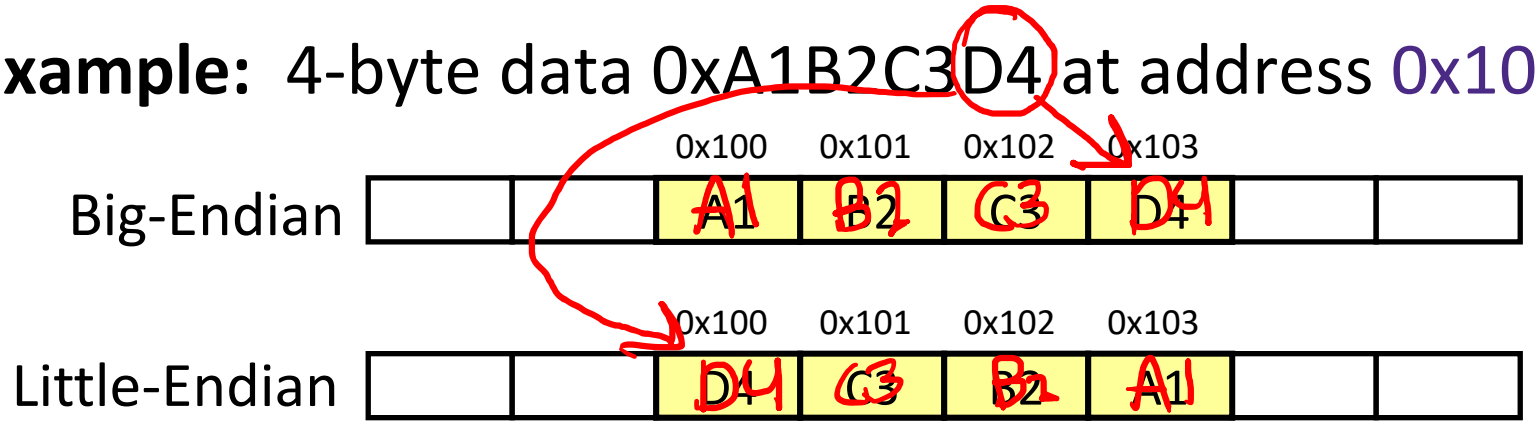
- How should bytes within a piece of data be ordered *in memory?*
  - Want to keep consecutive bytes in consecutive addresses
  - **Example:** store the 4-byte (32-bit) `int`:
    `0x A1 B2 C3 D4`

- By convention, ordering of bytes called *endianness*
  - The two options are big-endian and little-endian
    - In which address does the least significant **byte** go?
    - Based on *Gulliver's Travels*: tribes cut eggs on different sides (big, little)



Byte Ordering

- Big-endian (SPARC, z/Architecture)
  - Least significant byte has highest address (at the "big end")
- Little-endian (x86, x86-64)
  - Least significant byte has lowest address (at the "little end")
- Bi-endian (ARM, PowerPC)
  - Endianness can be specified as big or little
- **Example:** 4-byte data 0xA1B2C3D4 at address 0x100



Polling Question

Little endian
Least significant byte
Lowest address

- We store the value `0x 01 02 03 04` as a ***word*** at address 0x100 in a big-endian, 64-bit machine
- What is the ***byte of data*** stored at address 0x104?
  - Vote on Ed Lessons

A. 0x04

B. 0x40

C. 0x01

D. 0x10

E. We're lost…

0x100	0x101	0x102	0x103	0x104	0x105	0x106	0x107
00	00	00	00	01	02	03	04

Endianness

- *Endianness only applies to memory storage*
- Often programmer can ignore endianness because it is handled for you
  - Bytes wired into correct place when reading or storing from memory (hardware)
  - Compiler and assembler generate correct behavior (software)
- Endianness still shows up:
  - Logical issues: accessing different amount of data than how you stored it (*e.g.*, store `int`, access byte as a `char`)
  - Need to know exact values to debug memory errors
  - Manual translation to and from machine code (in 351)

Exploration Question

2) 0xD055C8 + 0x8
0x00D055D0

- Assume the state of memory is as shown below for a **little-endian** machine.

3) reverse byte order to preserve endianness

	0x100	0x101	0x102	0x103	0x104	0x105	0x106	0x107	
…	9F	23	B7	C8	55	D0	00	04	08	…
						D0	55	D0	00

1) 0x00D055C8 (little endian)

- If we (1) *read* the value of an `int` at address 0x102, (2) add 8 to it, and then (3) store the new value as an `int` at address 0x104, which of the following addresses retain their original value?

✓A. 0x102 ✓B. 0x104 ✗C. 0x105 ✗D. 0x107

Summary

- Memory is a long, *byte-addressed* array
  - Word size bounds the size of the *address space* and memory
  - Different data types use different number of bytes
  - Address of chunk of memory given by address of lowest byte in chunk
  - Object of K bytes is *aligned* if it has an address that is a multiple of K
- Pointers are data objects that hold addresses
- Endianness determines memory storage order for multi-byte data